Pascal–2 V2.1/RT-11 Language Specification

Introduction to the Language Specification

The Pascal–2 compiler processes the standard Pascal language, as described in the *Pascal User Manual and Report* [2nd edition], by Kathleen Jensen and Niklaus Wirth, published by Springer-Verlag, corrected printing of 1978. This language is more completely described in ISO Draft Proposal 7185, ISO/TC 97/SC 5, dated August 12, 1982, which we call the "draft standard" hereafter.

Compliance is Level 1: conformant array parameters are included. Pascal–2 includes the extensions detailed in this guide. This guide includes data on non-standard language features. This guide is not intended as a full language document.

Syntax definitions in this specification use the notation described in Appendix C, Pascal–2 Syntax.

Changes in the Standard

Because you may not be familiar with all the changes to the Pascal language from Jensen and Wirth (1978) to the most recent draft of the standard (1982), this section outlines those changes and Pascal–2's method of implementing them.

'For' Statement Control Variables

Variables that control a **for** statement must be simple variables, local to the routine in which the **for** statement is written. Originally, any variable could be used.

File Declaration

The standard states that the files **input** and **output** are automatically declared as global variables if they are mentioned in the program heading. Because program headings are optional in Pascal–2, **input** and **output** are declared as global variables in every Pascal–2 program. Thus, you cannot redefine **input** or **output** at the global level. In earlier versions of the language, the actual point of definition was undefined.

Parameter Compatibility

The compatibility rules for **var** parameters are now defined according to a restrictive rule, which requires the argument passed to have the same type as the formal parameter. Although the types must be the same, the type identifiers may differ. The appearance of a new type construct creates a new type. Previously, the rules for **var** parameters were undefined.

Procedure and Function Parameters

The draft standard has changed the method of declaring procedure and function parameters. The new syntax provides a way of checking the parameters of these procedures and functions, thus reducing the likelihood of type errors.

The syntax for a parameter list is changed to:

parameter-list = "(" *parameter-section* {";" *parameter-section* }")" .

parameter-section = (["**var**"] *identifier* {"," *identifier* }":" (*identifier*
| *conformant-array-schema*)) | *procedure-heading* | *function-heading* .

A full procedure heading must be provided for any procedure or function declared as a parameter, and the procedure heading for any procedure or function passed as an actual parameter must match. For example:

```
var
  K, L: integer;

  procedure P(procedure Q(I, J:integer));
  begin
    Q(K, L);
  end;

  procedure P1(I,J: integer);
  begin
    writeln('test of proc parameters', I, J);
  end;

begin
  K := 1;
  L := 20;
  P(P1);
end.
```

The program issues the following output:

```
test of proc parameters       1      20
```

The draft standard does not allow a standard function to be used as a parameter for a function or procedure. To pass a standard function as a function or procedure argument, you must define a function that calls the standard function, then pass the user-defined function as the function or procedure argument.

Conformant Array Parameters

Normally, a procedure or function accepts an array parameter containing a fixed number of elements. The number of elements holding meaningful information may vary but the size of the array may not. If you need to pass arrays of different lengths, you have to declare and pass a general array that is as long as the longest possible array, and you must track the last element of each. Another approach is to write a separate procedure to handle each size of array, which is clearly inefficient.

Use of conformant array parameters solves this problem. Conformant array parameters are formal parameters that allow you to write a general procedure or function that, at each activation, accepts array parameters of different size and with different lower and upper bounds. At activation, the upper and lower bounds of the conformant array parameter assume the upper and lower bounds of the passed parameter (the actual parameter).

The syntax for a conformant array parameter is:

conformant-array-parameter-specification =
["**var**"] *identifier-list* ":" *conformant-array-schema* .

conformant-array-schema = *packed-conformant-array-schema*
| *unpacked-conformant-array-schema* .

packed-conformant-array-schema = "**packed array**"
"[" *index-type-specification* "]" "**of**" *type-identifier* .

unpacked-conformant-array-schema = "**array**" "[" *index-type-specification*
{";" *index-type-specification* }"]" "**of**" (*type-identifier* | *conformant-array-schema*
) .

index-type-specification = *bound-identifier* ".." *bound-identifier* ":" *type-identifier* .

As the EBNF diagrams show, a conformant array schema may be either packed or unpacked. An unpacked conformant array may be nested within itself or within other conformant arrays (either packed or unpacked); if so, an abbreviated form may be used. In the example below, **Mx** is the conformant array parameter being used in **Examp**. **T1**, **T2** and **T3** are data types. The two definitions are equivalent. Notice that the semicolon in the abbreviated form replaces '] **of array** [' in the long form.

```
procedure Examp(var Mx: array [Lb1..Ub1: T1] of array [Lb2..Ub2: T2] of T3);
    or
procedure Examp(var Mx: array [Lb1..Ub1: T1; Lb2..Ub2: T2] of T3);
```

An array may be passed as a conformant array parameter if:

- the elements have the same types,
- the index types are compatible, and
- the bounds are within the range specified by the parameter declaration.

If two parameters are specified with a single conformant array schema, the actual parameter passed must have the same type. Also, a value conformant array may not be passed as a parameter to another procedure or function.

The next example demonstrates the use of conformant array parameters. The formal parameter **Arr** is a conformant array parameter and takes the values of two different-sized arrays, **First** and **Second**. At the first activation of the function **AddArray**, the two elements of array **First** are added together to reach a sum. The next activation adds up the four elements of array **Second** and arrives

at a different sum, as shown in the output following the program listing.

```
program Conform;
  var
    First: array [1..2] of integer;     { two-element array  }
    Second: array [0..3] of integer;    { four-element array }
    Total: integer;

  function AddArray(var Arr: array [Lower..Upper: integer] of integer): integer;
    var
      I, Sum: integer;
    begin
      Sum := 0;
      for I := Lower to Upper do
        Sum := Sum + Arr[I];
      AddArray := Sum
    end;

  begin
    First[1] := 5;  First[2] := 9;
    Total := AddArray(First);  ---------- called with two-element array
    writeln('Total for this array is: ', Total:5);
    Second[0] := 1;  Second[1] := -31;  Second[2] := 77;  Second[3] := 15;
    Total := AddArray(Second);  ---------- called with four-element array
    writeln('Total for this array is: ', Total:5)
  end.
```

Running the program yields:

```
Total for this array is:    14  ---------- sum of elements of array First
Total for this array is:    62  ---------- sum of elements of array Second
```

For a practical example of the use of conformant array parameters, see the source code of Pascal–2's Dynamic String Package, in the file STRING.PAS.

Literal Strings

A literal string may not extend over more than a single line. Earlier standards were unclear on this point. The limitation allows better diagnostics for unterminated strings.

'Write,' 'Writeln' of 'Packed Array of Char'

A **write** or **writeln** procedure call applied to a **packed array of char** writes only as many characters as the field-width parameter specifies. If the **packed array of char** exceeds the field-width, the string is truncated. The string is right-justified if the specified field width is longer than the packed array. If no field width is specified, a **write** or **writeln** writes as many characters as are in the string.

Example:

```
program Buff;
  var
    Buffer: packed array [1..30] of char;
    BuffCount: integer;

  begin
    Buffer:= 'This is a packed array of char';
    writeln(Buffer);
    BuffCount := 6;
    writeln(Buffer:BuffCount);
    writeln('cutoff':3);
    write('shorter':10);
  end.
```

When executed, the program yields these results:

```
This is a packed array of char
This i
cut
   shorter ———————————————————————————— note leading blanks
```

Identifiers

The initial character of an identifier must be an alphabetic character or a dollar sign. All other characters making up identifiers may be any combination of digits, letters, dollar signs or underbars. Identifiers may be of any length; all characters are significant. Lower-case characters are interpreted in the same way as upper-case characters. For example, **name**, **Name**, **NamE**, and **NAME** are equivalent. See "Syntax Extensions" for details on the the use of the non-standard dollar sign and underbar in identifiers.

Alternate Symbol Representations

The standard now defines alternate representations for symbols that are unavailable in some character sets. These are:

Standard Symbol	Alternate Symbol
^ or ↑	@ ('at' sign)
{	(*
}	*)
[	(.
]	.)

The alternate comment delimiters are equivalent to the standard comment delimiters, and a comment may open with one type of delimiter and close with the other. Comments may not be nested.

Examples:

```
(* This is a valid comment }
{ This is (* not *) a valid comment }
```

Implementation Definitions

This section provides details and characteristics of implementation-defined elements of Pascal–2.

Standard Type 'Integer'

The predefined identifier **maxint** has the value 32767.

The standard type **integer** has the range (-32767..32767). An unsigned (extended-range) integer may be defined with the range 0..65535. See "Unsigned Integer Conversion" in the Programmer's Guide.

Standard Type 'Real'

A **real** variable has the standard PDP-11 single-precision or double-precision floating-point structure, with magnitude in the range 1E-38..1E+38. Single-precision values give approximately 7 decimal digit precision; extended (double-precision) values give approximately 15-digit precision. Arithmetic overflow is detected for all **real** operations, but underflow is ignored and returns a result of zero.

The standard transcendental routines are accurate to 6 decimal digits in single precision and to 15 decimal digits in extended precision.

Standard Type 'Char'

The draft standard does not define the character set to be used internally to represent **char**. Pascal–2 uses 8-bit characters, allowing the use of the extended version of the ASCII character set, rather than 7-bit characters to represent the standard ASCII character set. The most significant bit is "off" unless used with extended character sets. **Ord(char)** is in the range 0..255.

Programs that calculate bit or byte offsets into a packed structure should treat a character as 8 bits, not 7; and storage size is the same for characters in either packed or unpacked structures.

Standard Type 'Text'

The standard type **text** is a file type with components of type **char**. **Text** is implemented as a file of 8-bit ASCII characters.

'Set' Types

Pascal–2 limits a **set** to a maximum of 256 elements. The lower and upper bounds must lie in the range 0..255, e.g., **set of 4..9**. The declaration **set of integer** is equivalent to the declaration **set of 0..255**. See "Undetected Errors" for restrictions on the checking of integer sets.

I/O Definitions

The following table summarizes the default field widths used when values are written to a text file:

Value Type	Field Width
`integer`	7
`real`	13
`boolean`	5

The floating-point representation of a real number includes the sign of the number (a space for positive numbers and a '-' for negative numbers), the real number in scientific notation, an uppercase E signifying exponential notation, the sign of the exponent ('+' or '-'), and a two-digit exponent. For example, the real number −105.39 prints as `-1.053900E+02`.

Boolean values are written in upper case (`TRUE`, `FALSE`). In the five-character default field, the value `TRUE` is right-justified, with a leading blank before the 'T'.

The procedure `page(F)` inserts a form feed (page eject) into the file specified by the required file argument. Calling `page(F)` with data in the file buffer executes `writeln(F)`, which writes the remainder of the buffer, and `write(F,chr(12))`, which writes the form-feed character. Calling `page(F)` with an empty file buffer results in a page eject only.

If associated with the standard input file (the terminal), `reset(input)` performs the equivalent of a `readln`, but otherwise has no effect; in the same way, `rewrite(output)` prints any incomplete line, but otherwise has no effect. `Reset(output)` or `rewrite(input)` produces an error message. See "External File Access" for details on the use of the extended form of `reset(input)` or `rewrite(output)`.

Syntax Extensions

This section describes Pascal–2 extensions to the syntax of standard Pascal.

Identifiers

The character **$** (dollar sign) is allowed in an identifier anywhere an alphabetic character is allowed. The character _ (underbar) is allowed anywhere a numeric character is allowed. For example, the identifier **_ABC** is not valid because it begins with an underbar. The following are legal identifiers:

```
system$name
$$file
this_is_a_long_identifier
This___Is_Also___Legal
```

Program Heading

In standard Pascal, the program heading is required, and the parameters define the external files to be used:

```
program Test (input, output, File3);
```

In Pascal–2, the program heading and parameters are not required. If present, they will be checked for proper syntax. The file parameters will otherwise be ignored. **Input** and **output** are automatically declared file variables. Every other external file must be specified by an additional parameter allowed in the standard procedures **reset** and **rewrite**. See "External File Access" under "I/O Support Extensions" for details.

Though not required, inclusion of the program name on the program statement is still a good practice because it names the object module for main programs and external modules. Further, the program name is used to name the psect when the **own** compilation switch is specified.

Declaration Order

The declaration sections **label**, **const**, **type**, **var**, **procedure**, and **function** may be interleaved as desired at the global level of a program. **Const** and **type** may be interleaved at other levels. This extension is useful for source module inclusion and structured constant definitions as described below. Any number of declaration sections of each type may be present. An identifier still must be defined before the identifier is used in any other way.

'%Include' Lexical Directive

A special directive allows separate text files to be included within a program. The contents of the separate file are inserted into the program at whatever point the **%include** directive occurs. Included files may themselves contain **%include** directives, nested to a maximum of seven levels.

The syntax for the **%include** directive is:

%include '*file-name-string*';

The *file-name-string* must contain at least the name of the file; if no file extension is specified, .PAS is assumed. In addition to the file name and extension, *file-name-string* may contain such information as the logical device name and disk volume number of the file.

The single quotes ('...') enclosing *file-name-string* are optional. This syntax provides compatibility with other implementations of Pascal–2 that allow file version numbers.

Examples:

```
%include hdr;
%include 'makhdr.pas';
%include tcrm.doc;
%include 'sy:hfil';
```

See the Programmer's Guide for details.

'%Page' Lexical Directive

The %page directive causes a page break (form feed) in the listing file immediately following the line on which the %page directive is placed. The %page directive itself is printed in the listing file on the last listed line of the page preceding the page eject. The ending semicolon is optional.

'External' and 'NonPascal' Directives

Similar to the **forward** standard directive, the **external** directive distinguishes a particular Pascal procedure or function that is separate from the module that invokes it. An external procedure must be declared at the global level. If the body of an external procedure or function does not appear in a compilation, it is assumed that the body will be in another object module. If the body of the external procedure does appear, its name will be made available in the object module for reference by other modules. References to the external procedure are resolved at link time.

Limitations of the object module structure require that external names be distinct within the first six characters. The underbar cannot be expressed in the object module format and is replaced by a period in the external name. No type checking is done for parameters of an external routine.

The **nonpascal** directive is used instead of **external** if the external procedure is written in a language other than Pascal. **Nonpascal** creates an interface between the Pascal–2 calling sequence of the program or module doing the calling and the DEC calling sequence required by non-Pascal external routine, usually a FORTRAN or MACRO-11 module. The **nonpascal** directive makes use of the convention of having register **R5** point to a list of parameters. All parameters are passed by reference, so only **var** parameters may be used. MACRO-11 routines written with the Pascal–2 PASMAC utility must be declared as **external** rather than **nonpascal**, because PASMAC simulates the Pascal–2 calling sequence.

See also "External Modules" in the Programmer's Guide for details.

Structured Constants

The syntax for constant definitions is extended to allow you to specify constants in record and array types. Under the standard, arrays or records cannot be assigned values in the constants declarations; each element must be assigned a value in the program body with an assignment statement. The structured-constants language extension eliminates the need to use assignment statements to assign values to constants of type **array** or **record**. See the examples following for a comparison of structured constants declarations versus standard constant declarations.

The formal syntax for structured constants is:

structured-constant =
structured-type-identifier constant-component-list .

constant-component-list = "(" *constant-component* {"," *constant-component* }")" .

constant-component = *constant* | *constant-component-list* .

where

structured-type-identifier
Is a data type with an array or record structure. All of the components of that structure must be of simple types, **array** types, or **record** types.

constant-component
Must correspond one to one with the component of the structured (array or record) type, and each *constant-component* must be a constant of the same type as the corresponding structure component. An access to the structure component returns the value of the *constant-component*. If the structure component is of a structured type, only the corresponding *constant-component-list* must be provided, declared with the proper syntax.

The following are valid declarations. Note that the data types needed by the structured constant must be declared before the structured constants.

```
type
  S1 = packed array [1..4] of char;
  S2 = record
    String: S1;
    end;

const
  C1 = S1('a', 'b', 'c', 'd');
  C2 = S2('abcd');
```

The *structured-type-identifier* for individual components need not be provided. For variant records (even those without a tag-field) a tag value must be provided in the *constant-component-list*.

Constants used as components in a *constant-component-list* appear between nested levels of parentheses. If an element is another structured type, a constant type of the same structure may appear or its elements may be set individually between inner parentheses. However, you may not use structured constants or their individual elements as **case** labels; **case** labels must be of simple type.

Examples of Structured Constants

Three examples are presented showing several uses of structured constants. The first example illustrates the nesting of parentheses in the structured constant declarations. The second example compares the standard's method of declaring record or array constants with the structured constant method. The third example shows the correct way to declare multidimensional arrays of constants.

The structured constant **Workers** in the following declarations contains three levels of parentheses: the first for the array structure; the next for the outer record; the last for the pay information. The fourth constant in the array, however, for Maxine, contains a structured element that is set by

reference to a constant of the same type with no further inner parentheses.

```
type
  Compensation = (Paid, Unpaid);
  Paytype = Record
    Title : (Clerk, Indian, Chief, President);
    case Compensation of
      Paid:   (Rate: real);
      Unpaid: ();
    end;
  Employeetable = array[1..4] of record
    Name : packed array[1..10] of char;
    Payinfo : Paytype;
    end;

const
  Conchief = Paytype(Chief, Paid, 6.85);   — note redefinition for Maxine
  Workers = Employeetable(
    ('Charlie   ', (Clerk, Paid, 3.40)),
    ('Samuel    ', (Indian, Paid, 5.25)),
    ('Edward    ', (President, Unpaid)),
    ('Maxine    ', Conchief)  ——————————— note condensed form
    );
```

To illustrate the efficiency and ease of use of structured constants, we present a comparison of the standard method of declaring constants for arrays and records versus the structured constants method. The program used in the comparison — **DayCalc** — calculates the day of the week for any date. The declarations below are those required to declare two arrays of constants, **MonthName** and **DayOffset**. Note that the type declarations are identical in both cases. The declaration of other data types, constants and variables have been omitted.

To conform to the standard, constants in arrays and records must be declared and assigned values as shown below. This method requires many more statements than the equivalent structured constants declarations, provided following the standard example.

```
program DayCalc;       { use of standard constant declarations }
  type
    Month = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec,
             Unknown);
    Name = packed array [1..3] of char;
    NameList = array [Month] of Name;
    DayOffsetList = packed array [Month] of 0..6;

  var
    MonthName: NameList;          { Text for name of month }
    DayOffset: DayOffsetList;     { Day mod 7 }
```

```
begin    { DayCalc }

  MonthName[Jan] := 'jan'; MonthName[Feb] := 'feb'; MonthName[Mar] := 'mar';
  MonthName[Apr] := 'apr'; MonthName[May] := 'may'; MonthName[Jun] := 'jun';
  MonthName[Jul] := 'jul'; MonthName[Aug] := 'aug'; MonthName[Sep] := 'sep';
  MonthName[Oct] := 'oct'; MonthName[Nov] := 'nov'; MonthName[Dec] := 'dec';
  MonthName[Unknown] := '???';

  DayOffset[Jan] := 0; DayOffset[Feb] := 3; DayOffset[Mar] := 3;
  DayOffset[Apr] := 6; DayOffset[May] := 1; DayOffset[Jun] := 4;
  DayOffset[Jul] := 6; DayOffset[Aug] := 2; DayOffset[Sep] := 5;
  DayOffset[Oct] := 0; DayOffset[Nov] := 3; DayOffset[Dec] := 5;
  DayOffset[Unknown] := 0;
            :  ------------------------------ rest of program goes here
```

With structured constants, on the other hand, your code is much shorter and easier to maintain than with the standard method. The only drawbacks are that the program is non-standard and is not necessarily portable to other Pascal implementations.

```
program DayCalc;       { use of structured constants }
  type
    Month = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec,
             Unknown);
    DayOffsetList = packed array [Month] of 0..6;
    Name = packed array [1..3] of char;
    NameList = array [Month] of Name;

  const
    MonthName = NameList('jan', 'feb', 'mar', 'apr', 'may', 'jun',
                         'jul', 'aug', 'sep', 'oct', 'nov', 'dec', '???');
    DayOffset = DayOffsetList(0, 3, 3, 6, 1, 4, 6, 2, 5, 0, 3, 5, 0);
              :  ------- rest of declarations and program body goes here
```

Multidimensional arrays of constants, such as the two-dimensional array below, can be declared as in the following program. This program prints the elements of the two-dimensional array `Table` in the order they are stored.

```
program TwoDimensions;
  const
    MaxElem = 3;

  type
    CharTable = packed array [1..MaxElem, 1..MaxElem] of char;

  const
    Table = CharTable(('x', 'y', 'z'),
                      ('a', 'b', 'c'),
                      ('p', 'd', 'q'));
  var
    I, J: integer;
```

```
begin     { TwoDimensions }
  for I := 1 to MaxElem do
    for J := 1 to MaxElem do
      write(Table[I,J], '  ');
  writeln;
end.      { TwoDimensions }
```

Running this program yields:

```
x  y  z  a  b  c  p  d  q
```

Default Case Label ('Otherwise')

A default statement can be included in a **case** statement according to the following syntax:

case-statement = "**case**" *case-index* "**of**" [*case-element* {";" *case-element* }]
[";"] ["**otherwise**" *default-statement* [";"]] "**end**" .

The default statement, which immediately follows the **otherwise** clause, is executed if no case label matches the value of the *case-index*. A special note on **otherwise** syntax: In contrast to the case label, which requires a colon, the **otherwise** clause must not contain a colon or a compilation error results.

Example:

```
case I of
  1: Ch := ':';
  9: Ch := ';';
otherwise Ch := '*';
end;
```

The list *case-element* is optional, so the following example is valid.

```
case I of
otherwise I := 1;
end;
```

I/O Support Extensions

I/O support extensions provide the Pascal–2 programmer with additional control of the interface to the operating system.

External File Access

The standard procedures **reset**, for opening an existing file, and **rewrite**, for creating a new file, have been extended to accept optional arguments that give Pascal–2 programs the ability to associate internal file variables with external file or device specifications. The syntax is as follows:

rewrite(*file-variable, device-or-file-name, default-values, file-status*);
reset(*file-variable, device-or-file-name, default-values, file-status*);

where

file-variable
: is a standard Pascal file variable.

device-or-file-name
: specifies the name of an external file with which the file variable is to be associated. This parameter, which may be a device or file name specification, must be a string type and may be either a literal string or a variable.

default-values
: is also of a string type, providing default values for any file fields not provided in the file name, including default file options.

file-status
: is an integer variable that is primarily used to return a special status code if the file cannot be opened; this code allows a program to recover from an otherwise fatal error. The fourth parameter also may be used to determine the number of blocks allocated to a file or to specify the number of blocks to allocated to a new file. These uses are explained in detail below.

Commas must separate any optional parameters used; a comma must be included to mark off an omitted parameter but need not follow the last included parameter. The following example opens a file for direct access and skips the file-name parameter, indicating a temporary file.

```
rewrite(F1, , '/seek');
```

See "Random Access to Data Files" for details on the **/seek** switch, and see "I/O Control Switches" in the Programmer's Guide for details on the use of other file switches.

The optional parameters may be used to redirect the standard files **input** or **output**, which by default are file variables associated with the standard terminal input or output devices, respectively. The next example redirects output from the terminal to the line printer.

```
rewrite(output, 'LP:');
```

Normally, an I/O error with **reset** or **rewrite** causes the support library to trap the error, terminate the program, and print an error message and procedure walkback. The fourth parameter may be used to return control to the program. If the fourth parameter is specified and an I/O error occurs, the support library sets the value of the fourth parameter to -1 and returns control to the program. You must check the value returned by the fourth parameter and specify what action to take if an error occurs. For example:

```
reset(infile, Filename, '.lst', status);   —— default extension of .LST
if status = -1 then UserProcessError       —— response needed to error status
else ContinueUserProgram;
```

Or you may use the predefined functions in the support library to initiate run-time diagnostics. These functions check the status of the fourth parameter and respond accordingly. See "Run-Time Error Reporting" in the Programmer's Guide. Either way, you must check the value of the fourth parameter each time you use it; otherwise, the program continues but may act unpredictably.

If the file is successfully opened, the fourth parameter returns the number of blocks allocated to the file. In addition, the fourth parameter may be used with **rewrite** to specify the number of blocks to be initially allocated to the file. When the size of the file is known in advance, this specification allows efficient space allocation by the operating system. If the file does not actually occupy the number of blocks specified, however, the operating system will truncate the file to the number of blocks needed. In turn, the value of the fourth parameter may be checked after a **rewrite** to be certain that the file was allocated the number of blocks you wished.

Two values for the fourth parameter have special meaning on RT-11. A value of 0 indicates that the file should be allocated one-half of the largest contiguous space. A value of -1 specifies that you want all contiguous space allocated to the file. In both cases, the value upon return is the amount of space actually given to the file or is -1 if an error occurred in opening the file. If the fourth parameter is absent, the file size is determined by the operating system and expands dynamically. Examples:

```
reset(f,'test', '.pas', size);  ——————————— assumes default of .PAS
writeln(size);  ——————————————————————— returns the size of the file in blocks
   :
size := 64;
rewrite(output, outstr, '.lis', size);  ——— file initially allocated 64 blocks
```

'Close' Procedure

The **close** predefined procedure indicates that its file parameter is no longer in use; **close** will reclaim buffer memory. Further access to the file is prohibited until **reset** or **rewrite** is used. Files are automatically closed upon program termination, or when they appear in another **reset** or **rewrite**; **close** allows files to be closed manually when it is necessary to reclaim buffer space before then. In addition, a file variable local to a procedure or function is automatically closed when that function or procedure terminates. See the sample program **Alphas** in the next section for implicit uses of **close**.

Random Access to Data Files ('Seek')

Pascal–2 includes the **seek** predefined procedure to allow direct access (random access) to data files opened with the **/seek** file control switch. The **seek** procedure requires two parameters: a file variable of the file to be accessed, declared as a **file of char** or other **file** type (but not of type **text**); and an integer record number (records in the file are numbered sequentially beginning with 1). After the **seek** call, the specified record is available in the file buffer if it exists; otherwise **eof** is set to indicate that the record is not available.

Seek also enables both reading and writing on the same file for in-place record updates. **Put** is required if the file buffer variable is to be written to the file. **Get** and **put** may be mixed with **seek** for sequential access, because the internal record pointer is updated after each **get** and **put**. See the example following for the use of **put** and **seek**.

After the file pointer is positioned by **seek**, both **read** and **write** as well as **get** and **put** may be performed. **Read** and **write** transfer data between the user variable and the file; **get** and **put** transfer data between the file buffer variable and the file. The following sequences may be used for direct access.

```
seek(F,I); read(F,V);  { read record I into V  }
seek(F,I); write(F,V); { write record I from V }
```

Example:

```
program Alphas;
  var
    C: char;
    F: file of char;

  begin
    rewrite(F, 'alpha.txt');        { open F for writing }
    for C := 'a' to 'z' do
      write(F, C);                  { write letters of the alphabet to F }
    reset(F, 'alpha.txt/seek');     { close and reopen F for seeking }
    seek(F, 4);                     { read record containing 'd' }
    writeln(F^);                    { write a 'd' to output }
    F^ := 'z';                      { 'd' becomes 'z' }
    put(F);                         { write 'z' to F in place of 'd' }
  end.  ------------------------------------------------ F closed automatically
```

As the program shows, the **/seek** file control switch must be used with **reset** or **rewrite** if the **seek** procedure is to be used to access the file. See "I/O Control Switches" in the Programmer's Guide for details.

At run-time, the character 'd' is written to the terminal. After program termination, the file ALPHA.TXT contains:

```
abczefghijklmnopqrstuvwxyz  ---------- z takes the place of d
```

Seek does not work on text files. For simulated random access on text files, you must use the **getpos** and **setpos** external procedures. See "Random Access to 'Text' Files" in the Programmer's Guide.

String Input ('Read' and 'Readln')

A character string is a **packed array [1..n] of char**. The **read** and **readln** procedures may be used to read variables of string types. Characters are read until the variable is filled. If **eoln** becomes **true**, the remainder of the string is filled with spaces. See "The Dynamic String Package" in the Utilities Guide for more sophisticated ways to read and manipulate strings.

'Break' Procedure

For efficiency, Pascal–2 buffers transmitted output data. **Break(F)** forces the actual transmission of data from a partially filled buffer of file **F**. This can be useful with interactive terminals or to guarantee actual transmission of data to a shared disk file.

Octal Output

In an integer **write** procedure call, a negative field-width specification will represent characters in octal (base 8).

Example:

```
write(I:-6);                { Display octal value of I }
```

Real Number Formatting

If the second formatting field is negative, a real number is printed in scientific notation. The number of digits to the right of the decimal point is the number specified in the second field. (The standard allows you to specify an integer constant or an integer expression in either formatting field.)

For example,

```
write(R:20:-5);
```

prints R with one digit to the left of the decimal point and five digits to the right, followed by an upper-case E, a sign character '+' or '-' and two digits signifying the exponent. The entire number is right-justified in a 20-character field.

If R has the value −367.2, the statement `writeln('R=',R:20:-5)` prints:

```
R=          -3.67200E+02
```

Low-Level Interface

This section describes Pascal–2 extensions that are useful to programmers needing access to machine-dependent characteristics.

Boolean Operators on Integer

The boolean operators **and**, **or**, and **not** may be applied to operands of **integer** or **integer** subrange type. The **not** operator is always applied first. The operators produce a 16-bit result of **integer** type.

Nondecimal Integer Constants

Nondecimal integer constants may be specified in two forms of notation. In the preferred form, the nondecimal value is written as shown:

nondecimal-integer-constant = *digit-sequence* "**#**" *hexadecimal-digit-sequence* .

where *digit-sequence* is the radix, or base, of the number, in the range 2..16. The number following the cross-hatch character '**#**' is any number represented in base *digit-sequence* notation. The '**#**' symbol is required regardless of base. For example, the decimal value 255 is written **8#377** for base 8 and **16#FF** for base 16. Also, the redundant form **10#255** is valid for the decimal value 255.

Pascal–2 supports another form of notation as a special case. Octal (base 8) notation for integer constants is signified by the suffix "**B**" (upper or lower case), so that **377B** and **377b** are the same value as 255 decimal.

Extended-Range Arithmetic

The normal range of **integer** variables in Pascal–2 is −32767..32767, but you also may declare **integer** types in the extended range of 0..65535. A variable with an upper limit greater than 32767 is called an extended-range or "unsigned" variable. Normal arithmetic operations, with the exception of division and modulo, are performed on extended-range variables. Comparisons and division are signed. An integer value may be assigned to an extended value, being converted as a bit pattern. If the value being assigned is negative, the error is not trapped at run-time, since there is no way for the compiler to tell the difference between a negative value and an extended-range value. The same sort of implicit transformation is true when an extended value is assigned to an integer variable. No conversion is done for constants.

The following sample program illustrates the way Pascal–2 handles extended-range numbers. Within the **repeat until** statement, the program reads an integer then prints it as an unsigned integer

and as a signed integer. The external procedure **Uwrite** is provided in the section on "Unsigned Integer Conversion" in the Programmer's Guide.

```
program BigNumberTest;
  type
    Unsigned = 0..65535;

  var
    BigNumber: Unsigned;

  procedure Uwrite(X: Unsigned; Width: integer);
    external;     { procedure to write an unsigned integer to output }

begin    { BigNumberTest }
  repeat
    write('Enter an integer: ');
    readln(BigNumber);
    write('  Unsigned, BIGNUMBER = ');
    uwrite(BigNumber,1);  writeln;
    writeln('  Signed, BIGNUMBER   = ', BigNumber:1);
    writeln;
  until false { forever };
end.     { BigNumberTest }
```

The program is executed, producing the following results. As mentioned earlier, the allowable range of values for the integer **BigNumber** is −32767..32767. The final entry — in fact, any value outside the range of possible integers — is an invalid value for an integer, halting the program with a walkback (unless **walkback** is disabled).

```
Enter an integer: -1
  Unsigned, BIGNUMBER = 65535
  Signed, BIGNUMBER   = -1

Enter an integer: -32767
  Unsigned, BIGNUMBER = 32769
  Signed, BIGNUMBER   = -32767

Enter an integer: 32767
  Unsigned, BIGNUMBER = 32767
  Signed, BIGNUMBER   = 32767

Enter an integer: -5555
  Unsigned, BIGNUMBER = 59981
  Signed, BIGNUMBER   = -5555

Enter an integer: 5555
  Unsigned, BIGNUMBER = 5555
  Signed, BIGNUMBER   = 5555

Enter an integer: 65535

PASCAL--I/O error at user PC= 1050B
Illegal value for integer

Error occurred at line 16 in program bignumbertest
```

See "Unsigned Integer Conversion" in the Programmer's Guide for Pascal routines that perform extended-range arithmetic and extended-range output.

"Origin" Declaration

A variable can be declared to have a particular address in the I/O page or system area with the following syntax:

var-declaration = *var-element* {"," *var-element*}":" *type* .

var-element = *identifier* ["origin" *constant*] .

The constant in the above syntax must have an integer value. A variable so specified has the address given by the integer following origin. This must be in the system space 0..777B or in the I/O page 160000B..177777B.

The following example demonstrates the use of origin, plus the use of the ref and size functions. See "Ref Function" and "Size and Bitsize Functions" for more details on those routines. The example controls a mythical device. The procedure ReadData sets up the device's control registers and initiates a transfer from the device into the task's memory. This example is specific to a machine without memory management hardware, such as a small RT-11 system.

```
program Device;                    { example of device control }

const
  Ready = 200B;                    { ready flag }
  ReadBuffer = 1;                  { read data command }

type
  Buffer = packed array [1..100] of char;
  BufferPointer = ^Buffer;

var
  StatusRegister  origin 177316B: integer;
  ControlRegister origin 177314B: integer;
  BufferAddress   origin 177312B: Buffer_Pointer;
  ByteCount       origin 177310B: integer;
  Data: Buffer;                    { holds data from device }

  procedure ReadData;
    begin    { ReadData }
      BufferAddress := ref(Data);          { Address for DMA xfer }
      ByteCount := size(Buffer);           { size of buffer }
      ControlRegister := ReadBuffer;       { start transfer }
      { Wait for device to complete transfer }
      while (StatusRegister and Ready) = 0 do {wait};
    end;    { ReadData }

begin    { Device }
  ReadData;
end.     { Device }
```

'Ref' Function

The **ref** function, with a variable argument of type T, produces a pointer to that variable with result type ^T (pointer to T). The **dispose** routine cannot always detect attempts to dispose of a pointer generated with this function, and you should not try to do so.

See the example under "Origin Declaration" and under "Loophole Function" for uses of **ref**.

'Size' and 'Bitsize' Functions

Two functions, **size** and **bitsize**, give the programmer information on the space allocated for values of different types. The functions have a single argument, a type identifier.

The function **size** returns the number of bytes that would be allocated for an object of that type by normal variable allocation. The function **bitsize** returns the number of bits that would be allocated for an object of that type as a component of a packed record. This is the actual number of bits required to hold the value.

For example, suppose you had declared a type **Subrange = 0..15** and called the functions **size** and **bitsize**, as in the following example program. The results tell you that two bytes and four bits are allocated for the argument in question.

```
program SizeBitsize;
type
  Subrange = 0..15;

begin
  writeln(size(Subrange));
  writeln(bitsize(Subrange));
end.
```

The program yields these results:

```
2 ———————————————————————— 2 bytes are allocated
4 ———————————————————————— 4 bits are allocated
```

These functions are primarily useful when you are interfacing with the operating system or with hardware functions.

See "Origin Declaration" for another example of **size**.

'Loophole' Function

The **loophole** function, by providing a controlled escape from Pascal type rules, allows you to assign variables of one type to a variable of a different type. One use, shown in the **DumpMemory** example following, is to convert a **pointer** type to an **integer** type, perhaps to perform pointer arithmetic. The Pascal–2 Debugger, which examines program data, uses **loophole** to look at the stack and compute the values of pointers.

The invocation of **loophole** requires two parameters:

loophole(*returned-type*, *expression-to-convert*);

where *returned-type* is an identifier specifying the data type to be returned by **loophole**, and *expression-to-convert* is an expression of a "compatible" type that is converted to *returned-type*. In this context two types are considered compatible only if they require the same amount of storage (see "Storage Allocation" in the Programmer's Guide), or if they are both non-real scalar types.

The result of the **loophole** function is the bit pattern of the second argument, expressed as a value of the type specified in the first argument.

The following program illustrates the compatibility rules that govern the use of **loophole**. The program coerces a **real** number to an equivalent two-word array of integers representing the two words used to store the real value, then coerces the two-word array back into a real number. The program then coerces an integer in the range 0..4 to a scalar of type **Car**, then coerces the scalar back to an integer. The **loophole(integer,S)** is equivalent to the statement **I := ord(S)**.

```
program Coerce;

  type
    Realequiv = array [0..1] of integer;
    Car = (Buick, VW, Datsun, Chevy, BMW); { scalar type }

  var
    Re: Realequiv;
    R: real;
    S: Car;   { scalar }
    I: integer;

  begin   { Coerce }
    write('Enter a Real number: ');
    readln(R);
    Re := loophole(Realequiv, R); { coerces real into 2-wd array of integers }
    writeln('Re = ', Re[0]:-8, Re[1]:-8);  { 2-wd array printed in octal }
    R := loophole(Real, Re);  { coerces 2-wd array back to real }
    writeln('R  = ', R);
    write('Enter an integer in range 0..4: ');
    readln(I);
    S := loophole(Car, I);    { coerces integer to scalar }
    write('S  = ');
    case S of     { writes the scalar value }
      Buick: writeln('Buick');
      VW: writeln('VW');
      Datsun: writeln('Datsun');
      Chevy: writeln('Chevy');
      BMW: writeln('BMW');
      end;  { case }
    I := loophole(integer, S); { coerces scalar back to integer }
    writeln('I  =', I);
  end.   { Coerce }
```

When executed, the program yields these results:

```
Enter a Real number: 21567.9
Re =    43650   77715    ———— octal representation of real number (2 words)
R  =  2.156790E+04
Enter an integer in range 0..4: 2
S  = Datsun
I  =      2
```

The only other method of type coercion is to declare a record with variants, using the fact that the compiler overlays storage for different variants. This method makes the same kind of assumptions

as the **loophole** function about the compiler's allocation of memory and machine's architecture. However, the **loophole** function has several advantages over variant records:

- No assumption need be made about field allocation in a variant record.
- The compiler checks that the different types are the same size.
- The bypassing of type checking rules is clearly marked (the compiler will flag **loophole** if the **$standard** switch is set). Also, if the code is used with a compiler other than Pascal-2, that compiler should mark **loophole** as an error, and appropriate changes can be made to the code. With variant records, the code might compile but not work.

The following sample program uses the **loophole** function to perform arithmetic on pointers so that a block of the task's memory can be printed.

```
program MDump;

  type
    Word = 0..65535;

    procedure DumpMemory(Start, Finish: Word);
      type
        Pointer = ^integer;
      var
        P: Pointer;

      begin    { Dump Memory }
        P := loophole(Pointer, Start);
        while loophole(Word, P) <= Finish do begin
          writeln(loophole(integer, P): -6, ': ', P^: -6);
          P := loophole(Pointer, loophole(Word, P) + 2);
          end;
      end;     { Dump Memory }

begin  { MDump }
  DumpMemory(1210B, 1220B);
end.   { MDump }
```

The program yields these results:

```
1210:      6
1212: 101032
1214:  10546
1216:  12746
1220: 177772
```

The next example shows a method to print the address of a variable of any type. The program creates a pointer to the variable, coerces the pointer type into the type used in procedure **WriteAddress**,

and prints out the address.

```
program PrintAddress;

  type
    U_Pointer = 0..65535;  { unsigned integers }

  var
    C: char;
    R: real;
    Cptr: ^char;
    Uptr: U_Pointer;
    P_Integer: ^integer;

  procedure WriteAddress(A: U_Pointer);
    begin
      writeln(A: -7); {octal value of address}
    end;

  begin   { PrintAddress }
    C := 'a';  R := 3.54;
    Cptr := ref(C); { create pointer to char }
    Uptr := loophole(U_Pointer, Cptr); { coerce pointer into an address }
    WriteAddress(Uptr);                { print out address }
    WriteAddress(loophole(U_Pointer, ref(R))); { print pointer to real }
    new(P_Integer);
    P_Integer^ := 3103;
    WriteAddress(loophole(U_Pointer, P_Integer)); {print pointer to integer }
  end.  { PrintAddress }
```

The program yields these results:

```
11106
11110
31436
```

Non-Standard Language Elements

Program Parameters

According to the standard, parameters supplied in the program header indicate external files. Further, the **input** and **output** files must appear in the program header if they are used in the program. The **input** and **output** files are always defined at the global level and may not be redeclared at that level.

With Pascal–2, the program header is not required, and any program parameters are entirely ignored (see "Program Heading" under "Syntax Extensions"). External files are referenced instead by an extended form of **reset** and **rewrite** using a second parameter (a string) giving the external filename (see "External File Access" under "I/O Support Extensions").

Directives

The draft standard treats standard directives such as **forward** as neither an identifier nor a reserved word. Pascal–2 treats the directives **forward**, **external**, **nonpascal** as reserved words. An identifier cannot have the same name as one of these directives.

'Mod' of Negative Numbers

The draft standard states that the divisor must be positive and the operator **mod** must have a non-negative result. That is,

$$0 \leq I \bmod J < J$$

The Pascal–2 compiler generates a divide instruction that gives a negative result if **I** is negative. The standard result can be generated by:

```
Result := I mod J;
if Result < 0 then Result := Result + J;
```

'Eof' Not Accurate For Binary Files

A RT-11 file structure is a sequence of 512-byte blocks. A file containing short records may actually end in the middle of a block, but no information is available as to the end of valid data in the last block, so the **eof** standard function should not be relied upon as accurate. Another method, such as a sentinel record or a record count, should be used to indicate the end of usable data.

Eof is correctly indicated for **text** files.

Structured Types as Function Return Values

Under the standard, functions can return simple data types only (e.g., **integer**, **real**, **char**). With Pascal–2, functions may return structured data types such as **record**, **array** and **set** types in addition to simple types. For example, the function **KeySort**, of structured type, is declared as:

```
function KeySort(Key: KeyType): StructType;
```

where *StructType* is the structured data type of the return value of **KeySort**.

Additional Predefined Functions and Procedures

The Pascal–2 system includes predefined functions and procedures, as allowed by the draft standard. Most of these are grouped according to function in other sections in this guide. This section describes miscellaneous predefined functions and procedures not otherwise described.

Because these procedures and functions are known to the compiler, they need not be declared in the program. The only exception is **timestamp**, which is functionally similar to the other procedures and functions but is not yet predefined. **timestamp** will be predefined in future releases but for now must be declared as an external procedure.

Procedure 'Delete'

The predefined **delete** procedure allows the deletion of a single file that is opened in a Pascal program. **Delete** accepts one argument, the file variable of the file to be deleted. Invoke the procedure with a statement similar to the following:

```
delete(F);
```

Internally, this procedure closes and deletes the file specified by the argument. Your program should not close the file (using **close**) before invoking the **delete** procedure. The run-time error message "can't delete file" results if the file cannot be deleted for some reason. See the example following the discussion of the **rename** procedure.

Procedure 'Rename'

The predefined procedure **rename** allows the renaming of an open file, from within a Pascal program. **Rename** accepts two arguments. The first argument passed to **rename** must be the file variable of the original file name. The second argument must be the file name of the new file. Invoke the procedure with a statement similar to the following.

```
rename(F, 'newfil.txt');  ——————————— renames F to NEWFIL.TXT
     or:
NewF := 'newfil.txt';
rename(F, NewF);  ——————————————————— renames F to NEWFIL.TXT
```

The second argument may be a constant, a variable, or a literal string. The second argument must contain at least one field. If any fields are omitted from the second argument, the omitted field takes the corresponding value from the original file name. For example, to change the extension only, use a statement similar to this:

```
rename(F, '.mac');  ———————— file name is the same; .MAC is the new extension
```

The original file must be open (via **reset**) before **rename** may be called on the file. The renamed file is automatically closed upon completion of the operation.

The following program illustrates the use of the **delete** and **rename** predefined procedures. The program reads a file of weather observations and weeds out duplicate reports, or "dupes." The

"good" reports are written to a file, which is later renamed. The file of duplicate reports is then deleted.

```
program Dupes;

  const
    Climat_File = 'climat.dat';

  var
    Data_File: text;          { file of weather observations }
    Dupe_File: text;          { file of duplicate reports }
    Good_File: text;          { file of good reports minus duplicates }

  procedure Discard_Dupes(var F, G, H: text);
    external;
      { This procedure sorts F, a file of weather observations,
        saving good reports on file G and discarding duplicate
        reports on file H. }

  begin      { Dupes }
    reset(Data_File, 'wear.dat');
    rewrite(Dupe_File, 'dupe.tmp');
    rewrite(Good_File, 'good.dat');
    Discard_Dupes(Data_File, Good_File, Dupe_File); { Weed out the dupes }
    rename(Good_File,Climat_File);   — renames GOOD.DAT to CLIMAT.DAT
    delete(Dupe_File);  ————————————— deletes the file of duplicates
  end.       { Dupes }
```

Predefined Function 'Time'

The predefined function **time** takes no parameters and returns a real value corresponding to the current time of day. The value **time** is represented in hours after midnight, so that 9:30 a.m. is 9.50 and 1:45 p.m. is 13.75. The resolution of **time** depends on the operating system, but all operating systems provide a resolution of at least one second.

The value returned could be used in header information. (If you wanted the date as well as the time, you would use **timestamp**, described below.) Or you could call **time** at the beginning and end of a text-processing program and write a procedure that calculates the number of lines processed per minute, based on the difference in value returned. Or, because it generates a real number, **time** may be used to "seed" a pseudo-random number generator. The example below returns uses **time** to return the time of day. **Chr(7)** is the "bell" character.

```
program WriteTime;

  var
    Hrs, Mins: integer;
    AmPm: packed array[1..2] of char;
```

```
begin     { WriteTime }
  Mins := Round(time * 60);
  Hrs := Mins div 60;
  Mins := Mins mod 60;
  if (Hrs < 12) then AmPm := 'AM'
  else if (Hrs = 12) and (Mins = 0)
    then AmPm := 'M ' else AmPm := 'PM';
  write('At the tone the time will be: ');
  write(((Hrs+11) mod 12 + 1):2);
  write(':', Mins div 10:1, Mins mod 10:1, AmPm:3);
  writeln(Chr(7));
end.      { WriteTime }
```

Running the program yields these results:

```
At the tone the time will be: 11:37 AM  <beep>
```

Procedure 'TimeStamp'

The **timestamp** procedure provides a way to obtain the date and time from within a Pascal program. Date and time are obtained simultaneously so that they are consistent, even close to midnight.

Timestamp is included in the Pascal–2 library, but the name is not pre-declared by the compiler. You must include a definition similar to:

```
procedure Timestamp(var day, month, year,              { date }
                    hour, min, sec: integer );         { time }
  external;
```

The following program prints the date and time using **timestamp**.

```
program DateTime(output);
  var
    Day, Month, Year: Integer;       { date data }
    Hour, Minute, Second: Integer;   { time data }

  procedure Timestamp(var Day, Month, Year,                  { date }
                      Hour, Min, Sec: Integer );             { time }
    external;

  procedure PrintTwo(N: Integer);
    begin { Print a number on the output file with two digits, including
            leading zeros if needed. The number must be 99 or less }
      write(output, N div 10: 1, N mod 10: 1);
    end; { PrintTwo }
```

```
begin     { DateTime }
  Timestamp(Day, Month, Year, Hour, Minute, Second);
  PrintTwo(Day);
  case Month of
     1: write(output, '-Jan-');
     2: write(output, '-Feb-');
     3: write(output, '-Mar-');
     4: write(output, '-Apr-');
     5: write(output, '-May-');
     6: write(output, '-Jun-');
     7: write(output, '-Jul-');
     8: write(output, '-Aug-');
     9: write(output, '-Sep-');
    10: write(output, '-Oct-');
    11: write(output, '-Nov-');
    12: write(output, '-Dec-');
    end;
  write(output, Year: 4, ' ');
  PrintTwo(Hour);
  write(output, ':');
  PrintTwo(Minute);
  write(output, ':');
  PrintTwo(Second);
  writeln(output);
end.      { DateTime }
```

The results of the program are:

```
16-Jun-1983 14:28:31
```

Error Handling

This section describes the errors defined by the Pascal standard and Pascal–2's handling of them.

Detected Errors

Pascal–2 detects the following errors in all cases:

1. **ln** or **sqrt** has a negative argument.
2. The integer value returned by **trunc** or **round** lies outside the range **-maxint..maxint**.
3. Integer or real division by zero.
4. The result of a real operation cannot be expressed because of limitations in the floating-point format.
5. No label matches the value of the case index in a **case** statement.
6. The characters being read from a text file do not represent a legal value for the type of variable being read.
7. An attempt to call **get**, **read**, or **readln** when the file has not been **reset** or when **eof** is **true** for that file.
8. An attempt to call **put**, **write**, **writeln**, or **page** when the file has not been rewritten or when **eof** is **false** for that file.
9. A call to **put** when the file variable is undefined.

Pascal–2 detects the following errors under these conditions:

1. The value assigned to a variable or value parameter is not within the declared range of values for that variable. Detected when the **$rangecheck** compiler switch is enabled. (Default.) Not detected when a negative value is assigned to an extended-range variable. See "Extended-Range Arithmetic" for more details.
2. An index expression for an array access is outside the range of the corresponding index type. Detected when the **$indexcheck** switch is enabled. (Default.)
3. A reference through a pointer with a **nil** or undefined value. Reference through a **nil** pointer is detected when the **$pointercheck** switch is enabled. (Default.) Reference through an undefined value is not detected, although many cases will be detected at compile time.
4. In a **for** statement, the initial and final values are not within the range of the controlled variable when the initial value is assigned to the controlled variable. Detected when the $rangecheck switch is enabled. (Default.)
5. The calling of **dispose** with a **nil** or undefined parameter. Detected if the parameter is **nil**; detected if the parameter was made undefined by a previous **dispose**. The **dispose** of an undefined pointer is sometimes detected.
6. The result of the **sqr** function is out of range. Detected if the argument type is **real**; undetected if the argument type is **integer**.
7. The result of **chr(x)** is not within the character set. Detected only if a value is assigned to a variable or is passed as a parameter.
8. The result of **succ** or **pred** lies outside the range of the type. Detected only if the value then is assigned to a variable or is passed as a parameter.
9. A **mod** with the right-hand side less than or equal to zero. Detected if the value is zero; otherwise not.

10. Reference to an undefined variable. Undetected in general. However, many simple cases are detected at compile time.
11. A return from a function without a value being assigned to the function. Undetected in general. However, many simple cases are detected at compile time.
12. An attempt to call **put** on a file that was opened with **reset**. Detected except for a file with the **/seek** file control switch specified when the file was opened.

Undetected Errors

Pascal-2 does not detect the following errors:

1. A set value assigned to a set variable or value parameter contains members not in the range of the base type of the set variable.
2. An access to a field in a variant record that is not selected by the current value of the tag-field.
3. A **dispose** of a variable allocated on the heap while there is an active reference to that variable as a variable parameter or in a **with** statement.
4. A change in the value of a file variable by a **get** or **put** while there is an active reference to that variable as a variable parameter or in a **with** statement.
5. Accessing of a variable allocated with **new**($p, c_1, ..., c_n$) as an entire variable, in an assignment or as a parameter.
6. Calling of **dispose(p)** when the value of $p^{\wedge}$ was created with **new**($p, c_1, ..., c_n$), or calling of **dispose**($p, c_1, ..., c_n$) with a variable created with **new** and a different set of tag values.
7. The result of an integer operation is incorrect because of overflow.
8. The value of a format expression to a **write** statement is less than 1. Undetected (used in a language extension).

Appendix A: Predefined Identifiers

Constants
- False
- Maxint
- True

Types
- Boolean
- Char
- Integer
- Real
- Text

Variables
- Input
- Output

Functions
- Abs
- Arctan
- Bitsize*
- Chr
- Cos
- Eof
- Eoln
- Exp
- Ioerror*
- Iostatus*
- Ln
- Loophole*
- Odd
- Ord
- Pred
- Ref*
- Round
- Sin
- Size*
- Sqr
- Sqrt
- Succ
- Time*
- Trunc

Procedures
- Break*
- Close*
- Delete*
- Dispose
- Get
- New
- Noioerror*
- Pack
- Page
- Put
- Read
- Readln
- Rename*
- Reset
- Rewrite
- Seek*
- Unpack
- Write
- Writeln

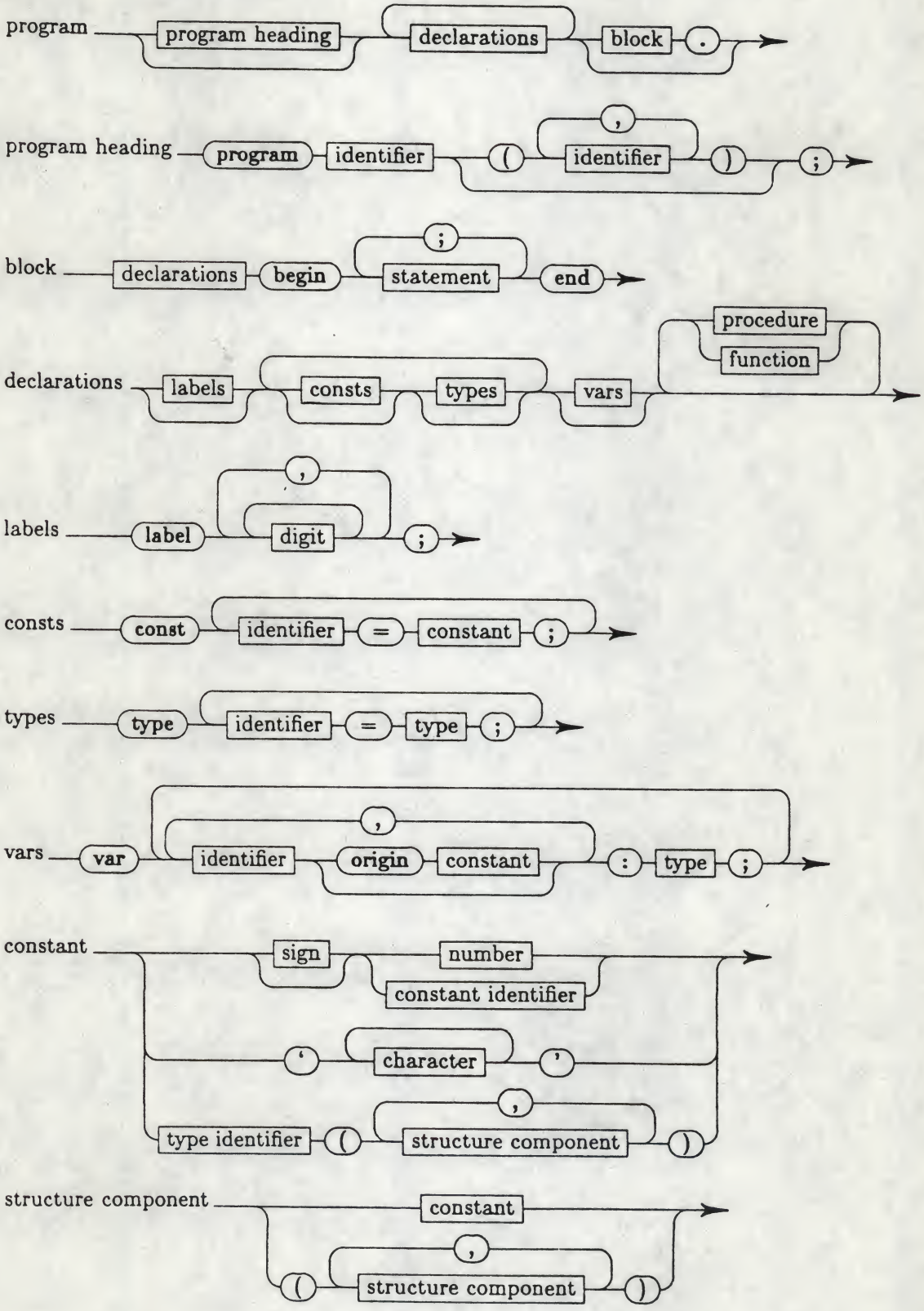
Appendix B: Reserved Words

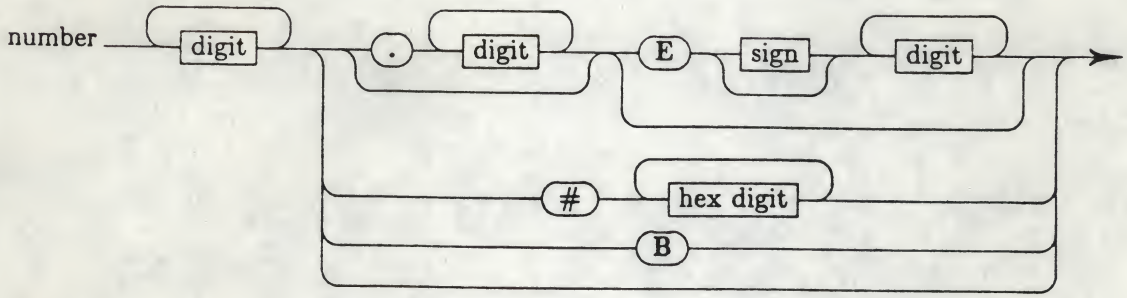
- And
- Array
- Begin
- Case
- Const
- Div
- Do
- Downto
- Else
- End
- External*
- File
- For
- Forward
- Function
- Goto
- If
- In
- Label
- Mod
- Nil
- Nonpascal*
- Not
- Of
- Or
- Origin*
- Otherwise*
- Packed
- Procedure
- Program
- Record
- Repeat
- Set
- Then
- To
- Type
- Until
- Var
- With
- While

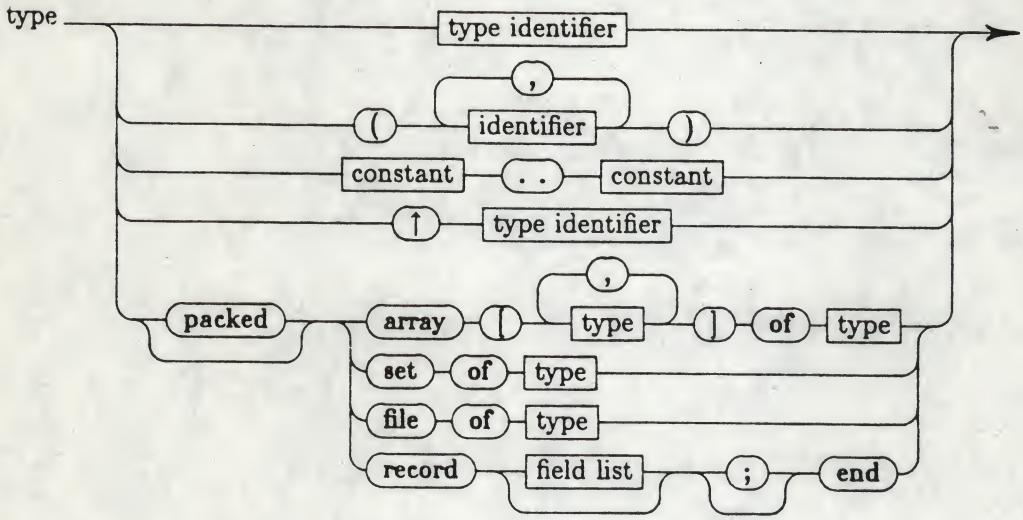
* Items marked with the asterisk are extensions of standard Pascal.

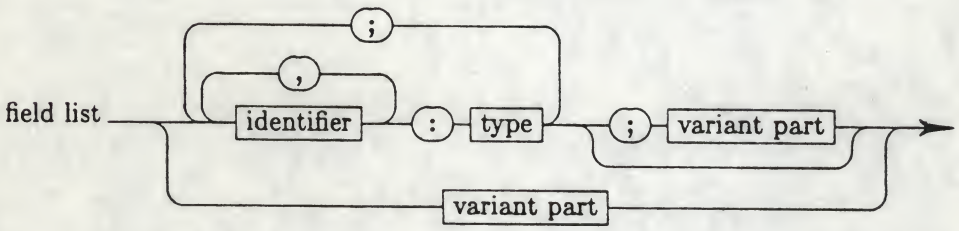
Appendix C: Pascal–2 Syntax

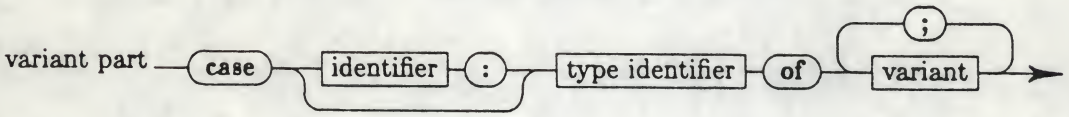
Pascal–2 Syntax Diagrams

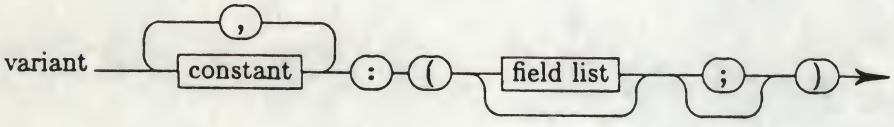


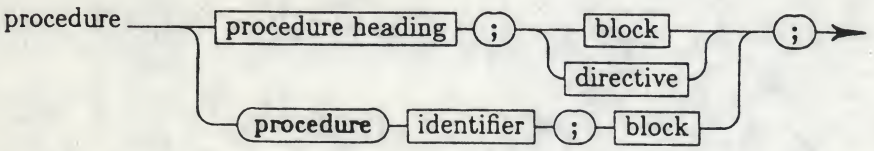

number
digit
.
digit
E
sign
digit
#
hex digit
B

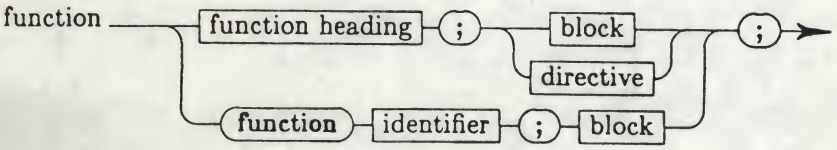

type
type identifier
(
,
identifier
)
constant
. .
constant
↑
type identifier
packed
array
[
,
type
]
of
type
set
of
type
file
of
type
record
field list
;
end

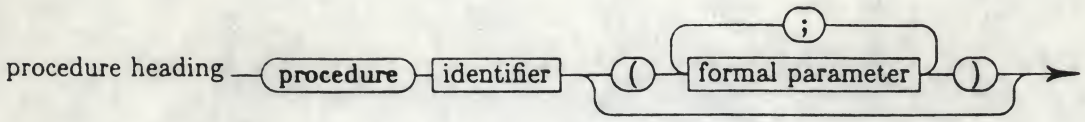

field list
;
,
identifier
:
type
;
variant part
variant part

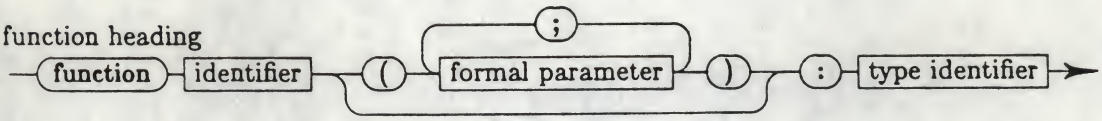

variant part
case
identifier
:
type identifier
of
;
variant

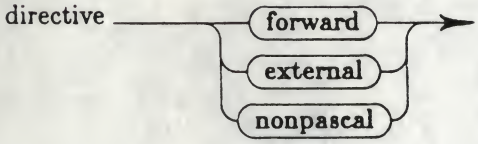

variant
,
constant
:
(
field list
;
)

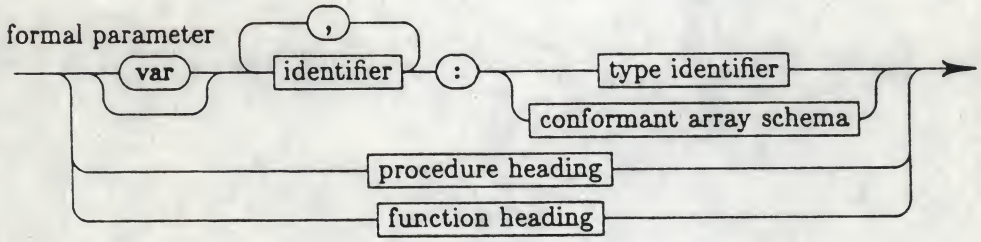

procedure
procedure heading
;
block
directive
;
procedure
identifier
;
block

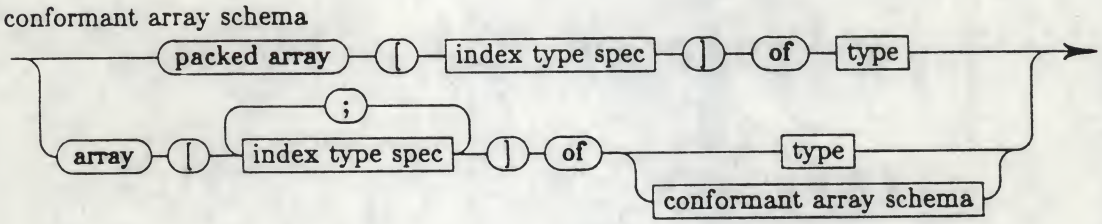

function
function heading
;
block
directive
;
function
identifier
;
block

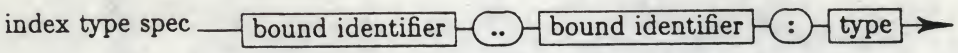

procedure heading
procedure
identifier
(
;
formal parameter
)

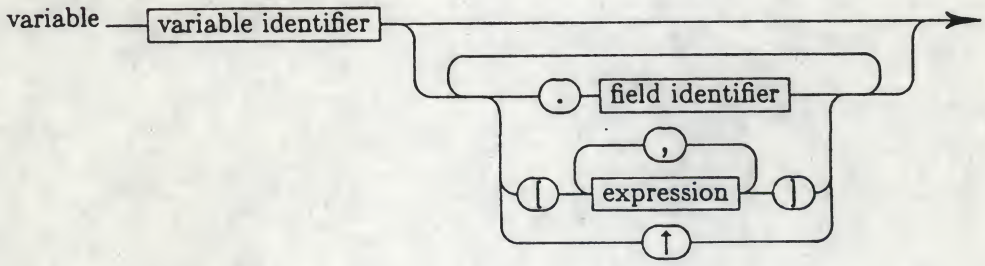

function heading
function
identifier
(
;
formal parameter
)
:
type identifier

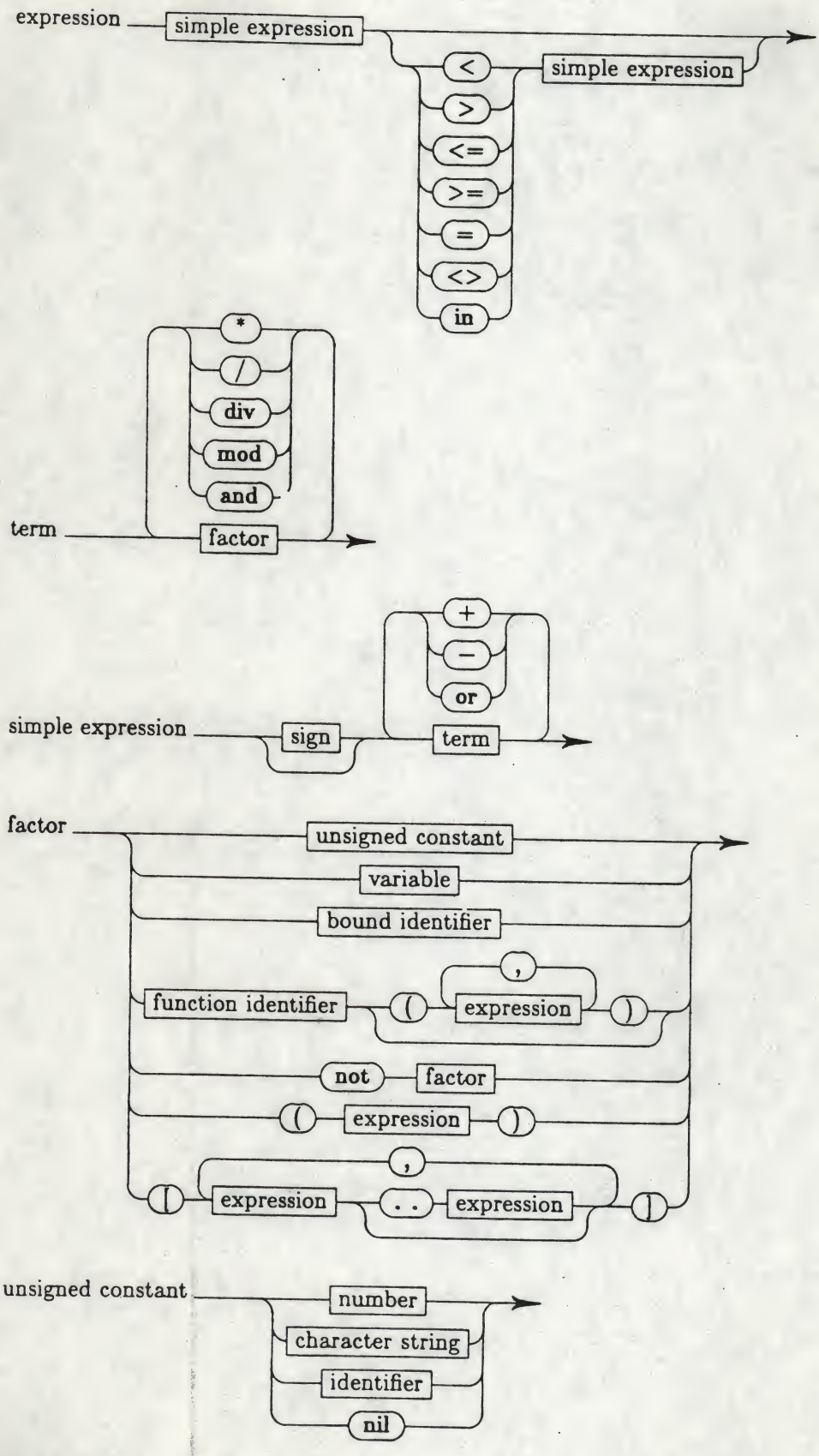

directive
forward
external
nonpascal

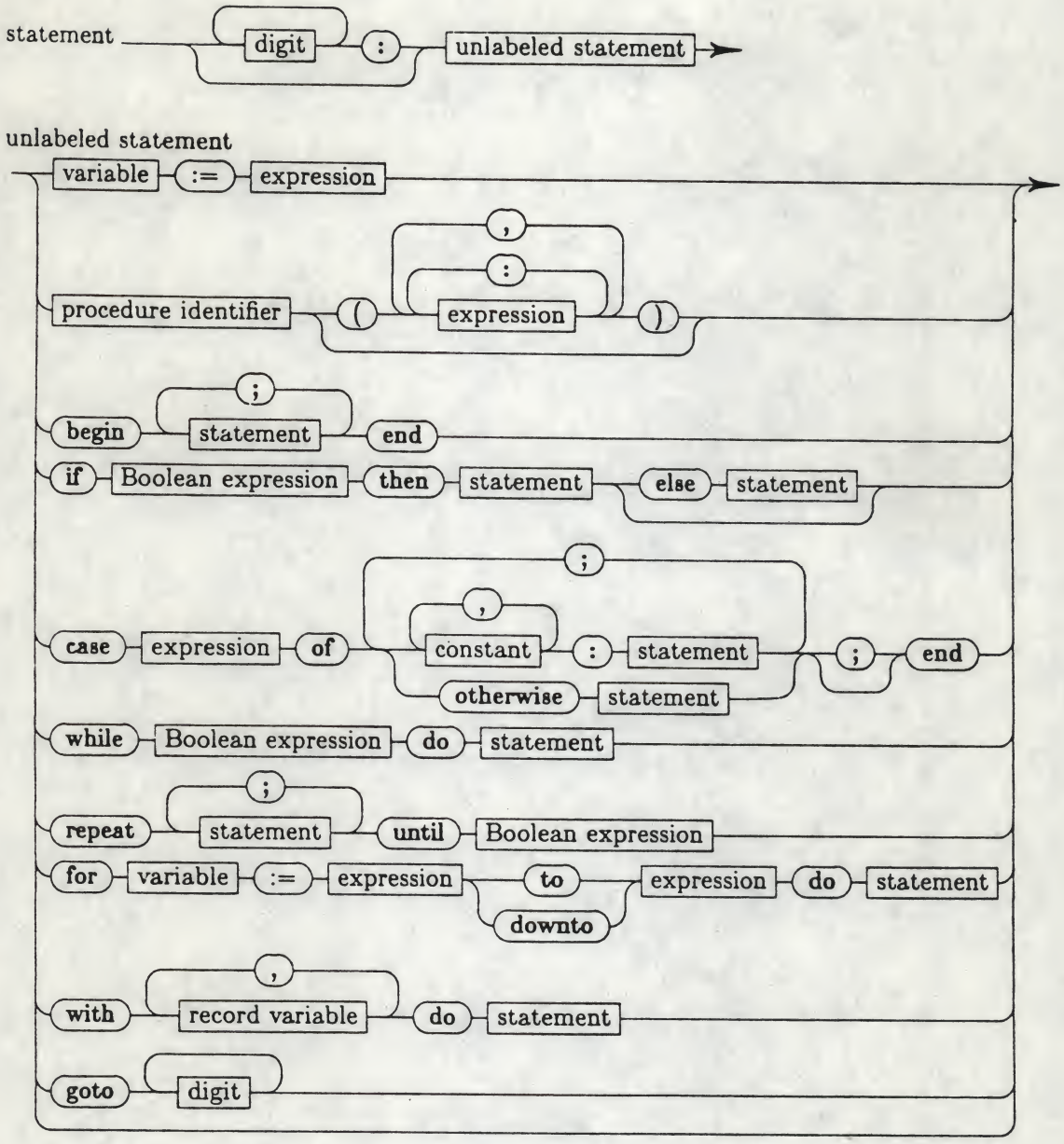

formal parameter
var
,
identifier
:
type identifier
conformant array schema
procedure heading
function heading


conformant array schema
packed array
[
index type spec
]
of
type
array
[
;
index type spec
]
of
type
conformant array schema


index type spec
bound identifier
..
bound identifier
:
type


variable
variable identifier
.
field identifier
,
[
expression
]
↑


expression
simple expression
<
>
<=
>=
=
<>
in
simple expression
term
*
/
div
mod
and
factor
simple expression
+
−
or
sign
term
factor
unsigned constant
variable
bound identifier
function identifier
(
,
expression
)
not
factor
(
expression
)
[
,
expression
..
expression
]
unsigned constant
number
character string
identifier
nil


statement
digit
:
unlabeled statement
unlabeled statement
variable
:=
expression
procedure identifier
(
,
:
expression
)
begin
;
statement
end
if
Boolean expression
then
statement
else
statement
case
expression
of
;
,
constant
:
statement
otherwise
statement
;
end
while
Boolean expression
do
statement
repeat
;
statement
until
Boolean expression
for
variable
:=
expression
to
downto
expression
do
statement
with
,
record variable
do
statement
goto
digit

Extended Backus-Naur Form

The notation used for describing syntax in this guide is a variant of the Backus-Naur Form (BNF) originally developed to describe the syntax of Algol 60. This particular variant was proposed by Niklaus Wirth ("What Can We Do About the Unnecessary Divergence of Notations for Syntactic Definitions?", *Communications of the ACM*, November 1977, vol. 20, number 11).

A "terminal symbol" is a symbol that actually appears in the language itself. Examples of terminal symbols in Pascal are:

```
begin   +   >=
```

Terminal symbols are written in quotes, e.g.: **"terminal"**.

Some terminal symbols are not easily expressed in this way, and these may be represented by comments contained in angle brackets <>. For example:

<any printable character>

A "nonterminal symbol" is used in the description of the language but does not actually appear in the text of the language. That is, it is used to talk about the language. A nonterminal symbol will stand for some sequence of terminal or nonterminal symbols. Nonterminal symbols are written without quotes. For example:

identifier, interface-part

A "production" is a rule specifying which terminal and nonterminal symbols make up another nonterminal symbol. A production is written:

left-hand-side = *right-hand-side* .

The *left-hand-side* is a nonterminal symbol; the *right-hand-side* is some combination of terminal and nonterminal symbols. A production indicates that the *left-hand-side* is made up of the symbols on the *right-hand-side*. A production is terminated with a period.

Within a right-hand-side, the following operators may occur:

(blank) indicates that the two symbols are concatenated. For example:

lhs = **"a" "b" "c"** .

indicates that *lhs* consists of the string **abc**.

| (vertical bar) indicates that the two symbols are alternatives. Concatenation is performed before alternation. For example:

lhs = **"ab"** | **"cd"** .

indicates that *lhs* consists of one of the strings **ab**, **cd**.

[ ] (brackets) indicate that the enclosed symbols are optional. For example:

lhs = **"a"** [**"bc"**] **"d"** .

indicates that *lhs* consists of one of the strings **abcd**, **ad**.

{} (braces) indicate that the enclosed symbols are repeated zero or more times. For example:

lhs = **"a"** {**"b"**}**"c"** .

indicates that *lhs* consists of any of ac, abc, abbc, abbbc, ...

( ) (parentheses) are used for grouping as they are in mathematics.

We can now use this notation to describe itself as an example. The productions for *letter*, *digit* and *character* are not given here but are obvious.

syntax = {*production* }.

production = *nonterminal-symbol* "=" *expression* "." .

expression = *term* {"|" *term* }.

term = *factor* {*factor* }.

factor = *nonterminal-symbol* | *terminal-symbol* | "(" *expression* ")"
| "[" *expression* "]" | "{" *expression* "}" .

terminal-symbol = "'" *character* {*character* }"'" | <*any comment in angle brackets*> .

nonterminal-symbol = *letter* {*letter* | *digit* | "-" }.

Pascal-2 Lexical Description

This set of productions defines the lexical representation of Pascal-2.

Productions that differ from the standard are marked with an asterisk (*).

The case of any alphabetic character is insignificant except in a *character-string*. Lower-case is used in this description.

1.* *letter* = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |"i"
| "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r"
| "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "$" .

2. *digit* = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .

3.* *octal-digit* = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" .

4.* *hexadecimal-digit* = *digit* | "a" | "b" | "c" | "d" | "e" | "f" .

5. *special-symbol* = "+" | "-" | "*" | "/" | "=" | "<" | ">" | "[" | "]" | "(." | ".)"
| "." | "," | ":" | ";" | "^" | "@" | "(" | ")" | "<>" | "<=" | ">="
| ":=" | ".." | *word-symbol* .

6.* *word-symbol* = "and" | "array" | "begin" | "case" | "const" | "div" | "do"
| "downto" | "else" | "end" | "file" | "for" | "function" | "goto" | "if"
| "in" | "label" | "mod" | "nil" | "not" | "of" | "or" | "origin" | "otherwise"
| "packed" | "procedure" | "program" | "record" | "repeat" | "set" | "then"
| "to" | "type" | "until" | "var" | "while" | "with" .

7.* *identifier* = *letter* {*letter* | *digit* | "_" }.

8. *bound-identifier* = *identifier* .

9.* *directive* = "forward" | "external" | "nonpascal" .

10. *digit-sequence* = *digit* {*digit* }.

11.* *unsigned-integer* = | *digit-sequence* "#" | *hexadecimal-digit-sequence* .

12. unsigned-real = (unsigned-integer "." digit-sequence ["E" scale-factor])
fi (unsigned-integer "E" scale-factor) .

13.* *nondecimal-integer* = *digit-sequence* "%"
(*hexadecimal-digit* {*hexadecimal-digit* }| *octal-number*) .

14.* *octal-number* = *octal-digit* {*octal-digit* }"b" .

15.* *unsigned-number* = *unsigned-integer* | *unsigned-real* | *octal-number* .

16. *scale-factor* = *signed-integer* .

17. *sign* = "+" | "-" .

18. *signed-integer* = [*sign*] *unsigned-integer;*

19. *signed-real* = [*sign*] *unsigned-real;*

20.* *signed-number* = *signed-integer* | *signed-real* | [*sign*] *octal-number* .

21. *label* = *unsigned-integer;*

22. *character-string* = "'" *string-element* {*string-element* }"'" .

23. *string-element* = "''" | <*any printable ASCII character*> .

24. *comment* = ("{" | "(*")
<*any sequence of characters and ends of lines not containing* "}" *or* "*)" >
("}" | "*)") .

25.* *lexical-directive* = "%include" "'" *file-name-string* "'" ";" | "%page" ";" .

Pascal–2 EBNF Syntax

This set of productions defines the syntax for the language accepted by the Pascal–2 compiler, including all extensions.

This section is to be interpreted in conjunction with the lexical description of the language.

Productions are based on those in the ISO standard. Where the language accepted by the Pascal–2 compiler differs from this standard, the production is marked with an asterisk ("*").

1.* *program* = [*program-heading*] {*label-declaration-part*
| *constant-definition-part* | *type-definition-part*
| *variable-declaration-part* | *routine-declaration* }[*body* "."] .

2. *program-heading* = "program" *identifier* ["(" *program-parameters* ")"] ";" .

3. *program-parameters* = *identifier* {"," *identifier* }.

4. *block* = *declarations body* .

5.* *declarations* = [*label-declaration-part*] [*constant-definition-part*]
[*type-definition-part*] [*variable-declaration-part*] {*routine-declaration* }.

6. *label-declaration-part* = "label" *label* {"," *label* }";" .

7. *constant-definition-part* = "const" *constant-definition* {";" *constant-definition* }";" .

8. *constant-definition* = *identifier* "=" *constant* .

9.* *constant* = ([*sign*] (*unsigned-number* | *identifier*))
| *character-string* | *structured-constant* .

10.* *structured-constant* = *structured-type-identifier constant-component-list* .

11.* *constant-component-list* = "(" *constant-component* {"," *constant-component* }")" .

12.* *constant-component* = *constant* | *constant-component-list* .

13. *type-definition-part* = "**type**" *type-definition* {";" *type-definition* }";" .

14. *type-definition* = *identifier* "=" *type* .

15. *type* = *identifier* | *enumerated-type* | *subrange-type* | *set-type*
| *array-type* | *record-type* | *file-type* | ("^" | "@" *identifier*) .

16. *enumerated-type* = "(" *identifier* {"," *identifier* }")" .

17. *subrange-type* = *constant* ".." *constant* .

18. *set-type* = ["**packed**"] "**set**" "**of**" *type* .

19. *array-type* = ["**packed**"] "**array**" "[" *type* {"," *type* }"]" "**of**" *type* .

20. *record-type* = ["**packed**"] "**record**" *field-list* [";"] "**end**" .

21. *field-list* = (*fixed-part* [";" *variant-part*]) | *variant-part* .

22. *fixed-part* = *record-section* {";" *record-section* }.

23. *record-section* = *identifier* {"," *identifier* }":" *type* .

24. *variant-part* = "**case**" [*identifier* ":"] *identifier* "**of**" *variant* {";" *variant* }.

25. *variant* = *constant* {"," *constant* }":" "(" [*field-list*] [";"] ")" .

26. *file-type* = ["**packed**"] "**file**" "**of**" *type* .

27. *variable-declaration-part* = "**var**" *variable-declaration* ";" {*variable-declaration* ";" }.

28. *variable-declaration* = *var-specification* {"," *var-specification* }":" *type* .

29. *var-specification* = *identifier* ["**origin**" *constant*] .

30. *routine-declaration* = (*procedure-declaration* | *function-declaration*) ";" .

31. *procedure-declaration* = (*procedure-heading* ";" *block*)
| (*procedure-heading* ";" *directive*) | (*procedure-ident* ";" *block*) .

32. *procedure-heading* = "**procedure**" *identifier* [*parameter-list*] .

33. *procedure-ident* = "**procedure**" *identifier* .

34. *function-declaration* = (*function-heading* ";" *block*)
| (*function-heading* ";" *directive*) | (*function-ident* ";" *block*) .

35. *function-heading* = "**function**" *identifier* [*parameter-list*] ":" *identifier* .

36. *function-ident* = "**function**" *identifier* .

37. *parameter-list* = "(" *parameter-section* {";" *parameter-section* }")" .

38. *parameter-section* = (["**var**"] *identifier* {"," *identifier* }":" (*identifier*
| *conformant-array-schema*)) | *procedure-heading* | *function-heading* .

39. *conformant-array-schema* = *packed-conformant-array-schema*
| *unpacked-conformant-array-schema* .

40. *packed-conformant-array-schema* = "**packed**" "**array**"
"[" *index-type-specification* "]" "**of**" *type* .

41. *unpacked-conformant-array-schema* = "**array**" "[" *index-type-specification*
{";" *index-type-specification* }"]" "**of**" (*type* | *conformant-array-schema*) .

42. *index-type-specification* = *bound-identifier* ".." *bound-identifier* ":" *type* .

43. *body* = *compound-statement* .

44. *statement* = [*label* ":"]
[*assignment* | *procedure-call* | *compound-statement* | *if-statement*
| *case-statement* | *while-statement* | *repeat-statement*
| *for-statement* | *with-statement* | *goto-statement*] .

45. *assignment* = *variable* ":=" *expression* .

46. *procedure-call* = *identifier* [*arg-list* | *write-arg-list*] .

47. *arg-list* = "(" *expression* {"," *expression* }")" .

48. *write-arg-list* = "(" *write-arg* {"," *write-arg* }")" .

49. *write-arg* = *expression* [":" *expression* [":" *expression*]] .

50. *compound-statement* = "**begin**" *statement* {";" *statement* }"**end**" .

51. *if-statement* = "**if**" *expression* "**then**" *statement* ["**else**" *statement*] .

52.* *case-statement* = "**case**" *expression* "**of**" [*case-element* {";" *case-element* }] [";"]
["**otherwise**" *statement* [";"]] "**end**" .

53. *case-element* = *constant* {"," *constant* }":" *statement* .

54. *while-statement* = "**while**" *expression* "**do**" *statement* .

55. *repeat-statement* = "**repeat**" *statement* {";" *statement* }"**until**" *expression* .

56. *for-statement* = "**for**" *identifier* ":=" *expression* ("**to**" | "**downto**") *expression*
"**do**" *statement* .

57. *with-statement* = "**with**" *expression* {"," *expression* }"**do**" *statement* .

58. *goto-statement* = "**goto**" *label* .

59. *expression* = *simple-expression* [*relational-operator simple-expression*] .

60. *relational-operator* = "**<**" | "**>**" | "**<=**" | "**>=**" | "**=**" | "**<>**" | "**in**" .

61. *simple-expression* = [*sign*] *term* {*adding-operator term* }.

62. *adding-operator* = "**+**" | "**-**" | "**or**" .

63. *term* = *factor* {*multiplying-operator factor* }.

64. *multiplying-operator* = "*****" | "**/**" | "**div**" | "**mod**" | "**and**" .

65. *factor* = *unsigned-constant* | *variable* | *function-call* | "**not**" *factor*
| "(" *expression* ")" | *bound-identifier* | ("[" | "(.")
[*member-designator* {"," *member-designator* }] ("]" | ".)") .

66. *unsigned-constant* = *unsigned-number* | *string* | *identifier* | "**nil**" .

67. *function-call* = *identifier* [*arg-list*] .

68. *variable* = *identifier* | *variable* ("[" | "(.") *expression* {"," *expression* }
("]" | ".)") | *variable* ("^" | "**@**") | *variable* "." *identifier* .

69. *member-designator* = *expression* [".." *expression*] .